# 基础知识

## 对象序列化

对象要被序列化，它的类必须要实现Serializable接口；

声明为static和transient类型的变量数据不能被序列化。因为static为类变量，transient代表对象的临时数据。

一个类实现了Serializable接口，那么它的所有子类都间接实现了此接口，所以它可以被序列化。

如果一个类中有引用类型的实例变量，这个引用类型也要实现Serializable接口。如Student类有Book类型的实例变量，要想让Student的对象序列化成功，则Book类必须实现Serializable接口。

能够对对象进行传输的貌似只有ObjectOutputStream和ObjectInputStream这些以Object开头的流对象。

为什么需要序列化？

我们的对象不只是存在内存中，还需要传输网络，或保存起来下次再加载出来用，所以需要Java序列化技术。

概念：

Java序列化是将对象转变成一串由二进制字节组成的数组，可以通过二进制数据保存到磁盘或传输网络，磁盘或网络接收者可以在对象的属类的模板上反序列化类的对象，达到对象持久化目的。

怎么序列化一个对象？

实现Java序列化接口：java.io.Serializable

序列化/反序列化工具？

借助common-lang3工具包实现序列化和反序列化。

byte[] bytes = SerializationUtils.serialize(user); // 序列化

User u = SerializationUtils.deserialize(bytes); // 反序列化

序列化注意事项：

序列化对象必须实现序列化接口。

序列化对象里面的属性是对象的话也要实现序列化接口。

类的对象序列化后，类的序列化ID不能轻易修改，不然反序列化会失败。

类的对象序列化后，类的属性有增加或者删除不会影响序列化，只是值会丢失。

如果父类序列化了，子类会继承父类的序列化，子类无需添加序列化接口。

如果父类没有序列化，子类序列化了，子类中的属性能正常序列化，但父类的属性会丢失，不能序列化。

serialVersionUID作用：

序列化时为了保持版本的兼容性，即在版本升级时反序列化仍保持对象的唯一性。

有两种生成方式：

-1）是默认的1L，比如：private static final long serialVersionUID = 1L;

-2）是根据类名、接口名、成员方法及属性等生成一个64位的哈希字段，比如：private static final long serialVersionUID = xxxxL;

序列化和反序列化？

序列化：把对象转换成字节序列的过程

反序列化：把字节序列恢复为对象的过程。

序列化描述：将数据结构或对象转换成能够被存储和传输（例如网络传输）的格式，同时应当要保证这个序列化结果在之后（可能是另一个计算环境中）能够被重建回原来的数据结构或对象。

对象的序列化用途？

1、把对象的字节序列永久地保存到硬盘上，通常存放一个文件中；

2、在网络上传送对象的字节序列。

当两个进程在进行远程通信时，彼此可以发送各种类型的数据。无论是何种类型的数据，都会以二进制序列的形式在网络上传送。发送方需要把这个Java对象转换为字节序列，才能在网络上传送；接收方则需要把字节序列再恢复为Java对象。

# 常量池（JDK1.8版）

相关链接：

https://blog.csdn.net/qq\_31615049/article/details/81611918

Java中的常量池分为三种类型：

-1）类文件中常量池（The Constant Pool）

-2）运行时常量池（The Run-Time Constant Pool）

-3）字符串常量池

常量定义：

final修饰，一旦赋值就无法改变！

final修饰的3种类型变量：静态变量、实例变量、局部变量

常量池：

主要存放两类常量：字面量（Literal）和符号引用（Symbolic References）。

字面量：相当于Java语言层面常量的概念，如文本字符串、声明位final的常量值等；

符号引用：属于编译原理方面的概念。包括如下三种类型的常量：类和接口的全限定名、字段名称和描述符、方法名称和描述符

## 类文件中常量池（Constant Pool）

---存在于Class文件中

诞生时间：编译时

内容概要：字面量（Literal）和符号引用（Symbolic Reference）

Class文件中存在一个常量池，存在的是编译期生成各种字面量和符号引用。

常量池指的是在编译期被确定，并被保存在已编译的.class文件中的一些数据。

字面量：包括文本字符串、被声明为final的常量值、基本数据类型的值。

符号引用：包括类和接口的全限定名、字段的名称和描述符、方法的名称和描述符。

常量池的每一项常量都是一个表，一共有11种各不相同的表结构数据，这每个表开始的第一位都是一个字节的标志位（取值1~12）。

Java中有8种基本类型和一个特殊类型String，这些类型为了使它们在运行过程中速度更快、更节省内存，都提供了一种常量池的概念。

## 运行时常量池

---存在于内存的元空间中

诞生时间：JVM运行时

内容概要：class文件元信息描述，**编译后的代码数据**，引用类型数据，类文件常量池。

运行时常量池：就是将编译后的类信息放入运行时的一个区域中，用来动态获取类信息。

运行时常量池是在类加载完成之后，将每个class常量池中的符号引用值转存到运行时常量池中，即每个class都有一个运行时常量池，类在解析之后，将符号引用替换成直接引用，与全局常量池中的引用值保持一致。

当java文件被编译成class文件之后，会生成上面的class常量池。而当类加载到内存之后，JVM就会将class常量池中的内容存放到运行时常量池中。

## 字符串常量池

--存在于堆中

字符串池中的内容是在类加载完成，经过验证、准备阶段之后，在堆中生成字符串对象实例，然后将该字符串对象实例的引用值存到String Pool中。（String Pool中存的是引用值而不是具体的实例对象）。

在HotSpot VM里实现的String Pool功能的是一个StringTable类，它是一个哈希表，里面存的是驻留字符串(也就是我们常说的用双引号括起来的)的引用（而不是驻留字符串实例本身）。

字符串各版本变化：

Jdk1.6及之前版本中，字符串常量池放在堆的Perm区，Perm区是一个类静态区域，主要存储一些加载类的信息、常量池、方法片段等内容，默认只有4m，一旦常量池使用大量intern是会直接产生java.lang.OutOfMemoryError:PermGen space错误的。

Jdk1.7版本中，字符串常量池从Perm区移动到堆Heap区，原因是Perm区大小了。

Jdk1.8版本中，已经取消Perm区，而新建立一个元数据区。

String类型的常量池主要使用方法有两种：

-1）直接使用双引号声明出来的String对象会直接存储在常量池中；

-2）若不是双引号声明的String对象，可使用intern()方法。

Intern()方法会从字符串常量池中查询当前字符串是否存在，若不存在则在堆内存创建当前字符串，再将其引用值存入常量池中并返回其引用值；若存在则直接返回其引用值。

字符串所在的内存区域：

String str1 = "helloworld"; //存在于常量池中

String str2 = new String("helloworld"); // 'helloworld'存在于常量池中，String对象存在于堆中

String str3 = str1 + "world"; // 有字符串引用时，在堆中建立一个String对象将引用返回

String ss = new String("abc"); //该语句会生成两个对象，一个是“abc”的实例对象，并且在StringTable中存储一个“abc”的引用值，还有一个是new出来的一个“abc”的实例对象，与上面那个是不同的实例。

# String类的使用

## String对象

大部分常量池中的字符串是在编译期确定的，除非很明确的调用String对象的intern方法返回（或创建）一个运行时常量池中的String对象。

(String s1 = "aa")中的这些String对象属于“interned String”。 String是Java对象，根据JVM规范它必须存在于java堆中，interned String也是。但JVM有个StringTable存着interned String的引用，保证内容相同的String对象不被重复intern。（这里便是编译器的优化）

字符串“+操作”：本质上是创建了StringBuilder对象进行append操作，然后将拼接后的stringBuilder对象用toString处理成String对象。

实例：

String s0 = "abc";

String s1 = "ab" + "c";

String s2 = "ab" + new String("c");

System.out.println(s0==s1); //true，说明：s0中的“abc”是字符串常量，在编译期就被确定了；而“ab”和“c”也是字符串常量，当一个字符串由多个字符串常量连接而成时，它自己肯定也是字符串常量，因此s1也是字符串常量，是“abc”的一个引用

System.out.println(s0==s2); //false，说明：s2后半部分无法在编译期确定，所以该字符串不放入常量池中

## intern()方法

源码：public native String intern();

定义：intern()方法首先会从常量池中查找是否存在该常量值，若存在则直接返回它的引用，若不存在则在常量池中创建后返回它的引用。

大体实现结构：Java使用jni调用c++实现的StringTable的intern()方法，StringTable的intern()方法跟Java中的HashTable差不多，只是不能扩容。若放进的String Pool的String非常多，就会造成Hash冲突严重，从而导致链表会很长，进而导致调用String.intern时性能会大幅下降。

jdk1.6中StringTable是固定的，大小为1009。

Jdk1.7可设置参数指定：-XX:StringTableSize=99991

Jdk1.7版本的String#intern()说明：

详情：https://www.cnblogs.com/wxgblogs/p/5635099.html

-1）将字符串常量池从Perm区移到了Java Heap区

-2）String#intern方法时，若若存在堆中的String对象，字符串常量池会保存堆中String对象的引用，而不是重新创建对象。

实例一：（jdk1.7版本）

String s3 = new String("1") + new String("1");

s3.intern();

String s4 = "11";

System.out.println(s3 == s4); //true

实例二：（jdk1.7版本）

String s3 = new String("1") + new String("1");

String s4 = "11";

s3.intern();

System.out.println(s3 == s4); //false

实例三：（jdk1.7版本）

String s3 = new String("1") + new String("1");

String s4 = "11";

s3 = s3.intern();

System.out.println(s3 == s4); //true

## String、StringBuffer、StringBuilder区别

-1）String是字符串常量，StringBuffer和StringBuilder都是字符串变量。

-2）StringBuffer是线程安全的，而StringBuilder是非线程安全的。由于线程安全会带来额外的系统开销，所以StringBuilder的效率比StringBuffer高

## replace、replaceAll区别：

相同点：都是全部替换，如果仅替换一次可使用replaceFirst()方法。

1）replace参数是char和CharSequence，即支持字符替换和字符串替换两种。

2）replaceAll的参数是regex，即基于规则表达式的替换。如replaceAll("\\d", "\*")把字符串中所有数字字符换成\*号

## indexof()和lastIndexOf()区别

indexOf(str [, startIndex]) //从左到右执行查找

lastIndexOf(str [, startIndex]) //从右到左执行查找

实例：查找” bbb”最近的”/\*”和”\*/”

String str = "/\* aaa \*/ /\* bbb \*/ /\* ccc \*/";

int index = str.indexOf("bbb");

int indexOf = str.indexOf("\*/", index);

int lastIndexOf = str.lastIndexOf("/\*", index);

System.out.println(indexOf + " " + lastIndexOf); //输出: 18 11

## 字符串格式化

String.format("%05d", 2); //输出00002，位数不变前面补零

MessageFormat.format("{0} + {1} = {2}", 1, 2, 1+2) //输出1 + 2 = 3

# 抽象类/接口

抽象类：

JDK1.8以前，抽象类方法默认访问权限为protected；

JDK1.8以后，抽象类方法默认访问权限为default；

接口：

JDK1.8以前，接口中的方法必须为public；

JDK1.8时，接口中的方法也可以是default；

JDK1.9时，接口中的方法也可以是private；

抽象类和接口的区别：

-1）抽象类中可存在非抽象方法；接口中的方法必须是抽象方法。

-2）抽象类中可以有普通的成员变量；接口只有常量，没有变量，必须是static final类型的，必须初始化（原因：类加载时期加载static参数）。

-3）Java8中接口会有default方法，即方法可以被实现。

-4）每个子类的构造函数都在第一行默认调用super()方法，new子类时会调用父类的构造方法，这是接口没有的。详情https://www.cnblogs.com/anrainie/archive/2012/03/28/2420738.html

选择：

-1）优先选择接口！

-2）若存在某个具体实现的方法（如BaseDao类注入了SessionFactory），只要方法没有全部实现，则设置成抽象类。

# 异常

try-catch-finally规则：

-1）try之后添加catch块或finally块。

-2）一个 try 块可能有多个 catch 块，但只会匹配一个catch块。若匹配一个匹配块，就不会再执行其它的catch代码块了。

-3）当try块和finally块都有return返回时，该方法返回的是finally的。但不建议在finally块写return语句。

-4）当try块中有System.exit(0)，表示整个虚拟机里的内容都释放，JVM停止工作，程序正常退出。此时finally中的语句不会执行。

try块和finally块的return语句描述：

try中执行到return时，先把这个值存起来，再开辟一块内存存这个值，然后去执行finally，finally执行之后，回去执行之前没执行完的return语句，将值返回。

throw与throws区别：

-1）throw是抛出一个异常对象。（如：throw new Exception()）

-2）throws是方法上声明可能抛出的异常们，表示抛出的异常由调用该方法的调用者处理。（如：public void saveAll() throws Exception1,Exception2{ … }）

实例：

public void doA(int a) throws Exception1, Exception3 {

try {

...//省略

}catch(Exception2 e2) {

System.out.println("捕获并处理Exception2异常！");

}

if(a!=b) throw new Exception3();

}

说明：

1）如果产生Exception1异常，则捕获之后再抛出，由该方法的调用者去处理。

2）如果产生Exception2异常，则该方法自己处理了(即System.out.println("捕获并处理Exception2异常！");)。所以该方法就不会再向外抛出Exception2异常，public void doA(int a) throws Exception1,Exception3里面的Exception2也就不用写了。

3）Exception3异常是该方法的某逻辑出错，程序员自己做了处理，在该段逻辑错误的情况下抛出Exception3异常，则该方法的调用者也要处理此异常。

# 内部类

4种内部类：成员内部类、静态内部类、局部内部类和匿名内部类

为什么需要内部类：

-1）高内聚，把只和这个类相关的类型放到这个类的内部。例如链表、接口回调。

-2）设计原则中的“组合优先于继承”：若继承仅仅是为了得到父类的功能，则此时应该完全抛弃继承，而改用组合实现。

实例：比较器类MyComparator、HashMap集合中的节点Entry

## 成员内部类

格式：

public class Outer {

class Inner {...} //成员内部类

}

创建内部类对象：Outer.Inner inner = (new Outer()).new Inner();

访问外部类非静态变量：外部类名.this.变量名

访问外部类静态变量：外部类名.变量名

说明：

相当于外部类的一个成员方法，访问权限与成员方法一样，能能访问外部类的所有变量和方法，包括静态和非静态，私有和非私有。

注：成员内部类没有静态变量和静态方法！！

注：内部类是一个编译时的概念，一旦编译成功，就会成为完全不同的两类。编译完成后出现outer.class和outer$inner.class两类。Out.java编译后会生成两个class文件，分别是Out.class和Out$Inner.class。

好处：用内部类定义在外部类中不可访问的属性。这样就在外部类中实现比外部类的private还小的权限。

外部类访问内部类：

注：成员内部类没有静态变量和静态方法！

--外部类的非静态方法访问成员内部类的非静态方法：

public void outer\_f3() {

new Inner().inner\_f1();

}

--外部类的静态方法访问成员内部类的非静态方法：

public static void outer\_f4() {

new Outer().new Inner().inner\_f1();

}

## 静态内部类

格式：

public class Outer {

static class Inner{...} //静态内部类

}

创建静态内部类对象：Outer.Inner in = new Outer.Inner();

访问静态内部类的静态成员：String staticStr = Inner.staticStr;

访问静态内部类的非静态成员：String str = new Outer.Inner().str;

说明：

用static修饰。访问权限与外部类的静态方法一样，只能访问外部类的静态变量和静态方法。

注：Outer.java文件编译后会生成两个class文件，分别为Outer.class和Outer$Inner.class。

应用场景：

1）当类与接口（或接口与接口）发生方法名冲突时，必须使用内部类来实现。用接口不能完全地实现多继承，用接口配合内部类才能实现真正的多继承。

2）Java集合类HashMap内部就有一个静态内部类Entry。Entry用来存放元素，HashMap的主干为一个Entry数组。

静态内部类与成员内部类的区别：

-1）内部：静态内部类中可以定义静态成员，而成员内部类中不允许定义静态成员。

-2）外部：静态内部类只能访问外部类的静态变量和静态方法；而成员内部类可以访问外部类所有的变量和方法，包括静态和非静态，私有和非私有。

访问：

class OuterOne {

public class InsideOne {...} //成员内部类

public stitic class InsideTwo{...} //静态内部类

public static void main(String[] args) {

OuterOne.InsideOne one = new OuterOne().new InsideOne(); //成员内部类访问

OuterOne.InsideTwo two = new OuterOne.InsideTwo(); //静态内部类访问

}

}

## 局部内部类

即在方法中定义的内部类，与局部变量类似，不能用public或private修饰。

局部内部类中不可定义静态变量。

定义在实例方法中的局部类可以访问外部类的所有变量和方法，定义在静态方法中的局部类只能访问外部类的静态变量和方法。

局部内部类只能访问final类型的局部变量。

格式：

public class Outer { //外部类

public void f() { //方法

class Inner{...} //局部内部类：定义在方法内

}

}

## 匿名内部类

格式：new 接口()|父类构造器(参数列表) { ...//匿名内部类的类实现 }

是一种特殊的局部内部类，使用匿名内部类时，必须继承一个类或实现一个接口。（如创建对象：Date d = new Date(){...}）

由于构造器的名字必须与类名相同，而匿名类没有类名，所以匿名类不能有构造器。

匿名内部类中不能含有静态成员变量和静态方法。final修饰的局部变量才可被匿名内部类使用。

匿名内部类不能是public、protected、private、static

应用场景：

匿名内部类使用广泛，比如我们常用的绑定监听的时候。

# BeanUtils/Properties工具类

org.apache.commons.beanutils.BeanUtils.copyProperties(user, user2);

org.apache.commons.beanutils.PropertyUtils.copyProperties(user1, user2);

org.springframework.beans.BeanUtils.copyProperties(attach, ebankApply4);

--1）get方法

BeanUtils.getProperty(userBean,"username");

BeanUtils.getProperty(studentBean, "class.name"); //获取学生对应的班级名称

BeanUtils.getProperty(orderBean, "customers[1].name"); //获取list的属性

--2）set方法

BeanUtils.setProperty(p, "name", "张三");

--3）populate()方法

在MVC框架中，经常要从request，resultSet等对象取出值来赋入bean中。

若不想写：String a = request.getParameter(“a”); bean.setA(a); String b = …

不妨写一个Binder：

MyBean bean = ...;

HashMap map = new HashMap();

Enumeration names = request.getParameterNames();

while (names.hasMoreElements()) {

String name = (String) names.nextElement();

map.put(name, request.getParameterValues(name));

}

//支持java.sql.Date和java.sql.Timestamp两种类型

BeanUtils.populate(bean, map);

Servlet中有这样的使用：

Person person = new Person(); //这是一个JavaBean

BeanUtils.populateBean(person, request);

---->populateBean(person, request.getParameterMap()); //先将request内容转为Map类型

---->BeanUtils.populate(info, propertyMap);

-4）PropertyDescriptor类

作用：操作访问JavaBean的属性。

# java.beans.PropertyDescriptor.PropertyDescriptor(String, Class<?>)

-1）操作一个属性(get/set使用)

Person obj = new Person();

PropertyDescriptor pd = new PropertyDescriptor("username", Person.class); //得到对象的属性

Method method = pd.getWriterMethod(); //获取setter方法

method.invoke(obj, "testSetMethod");

Method method = pd.getReaderMethod(); //获取getter方法

Object value = method.invoke(obj);

-2）操作多个属性（常用！！）

Person bean = new Person();

PropertyDescriptor[] propertyDescriptors = PropertyUtils.getPropertyDescriptors(bean);

for (PropertyDescriptor propertyDescriptor : propertyDescriptors) {

String propertyName = propertyDescriptor.getName();

Object value = PropertyUtils.getProperty(bean, propertyName);

}

# 配置文件处理类(Properties)

--Properties类是HashTable的子类，该类用于处理配置文件。

实例：

Properties property = new Properties();

InputStream input = ClassLoader.getSystemResourceAsStream("db.properties");

property.load(in);

String user = property. getProperties(“user”); //使用一

Set<String> propertyNames = property.stringPropertyNames(); //遍历一

for(String name : propertyNames) {

property.getProperty(name);

}

Map<String, String> map = new HashMap<String, String>((Map)property); //遍历二

for(Map.Entry<String, String> entry : map.entrySet()) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}